MERIDIAN
SOFTWARE SYSTEMS

Meridian Ada 4.1

Math Library

Preface

```
Proposed Standard for a                 -- not the official copy, ASCII text
Generic Package of                      -- differs from type set proposed
Elementary Functions for Ada            -- standard, yet, this is useful for
ISO-IEC/JTC1/SC22/WG9 (Ada)             -- information purposes
Numerics Rapporteur Group

Draft 1.1
3 October 1989
```

This document defines the specification of a generic package of elementary functions called GENERIC_ELEMENTARY_FUNCTIONS and the specification of a package of related exceptions called ELEMENTARY_FUNCTIONS_EXCEPTIONS. It does not define the body of the former. No body is required for the latter.

Deriving its content from a joint proposal of the ACM SIGAda Numerics Working Group and the Ada–Europe Numerics Working Group, the WG9 Numerics Rapporteur Group submitted Draft 1.0 of this proposed standard to WG9 in March, 1989. WG9 approved Draft 1.0 in June, 1989, subject to a revision of Section and a number of editorial improvements; these changes have resulted in this current version, Draft 1.1, which is submitted for consideration as an international standard.

The generic package described here is intended to provide the basic mathematical routines from which portable, reusable applications can be built. The standard serves a broad class of applications with reasonable ease of use, while demanding implementations that are of high quality, capable of validation, and also practical given the state of the art.

The two specifications included in this document are presented as compilable Ada specifications followed by explanatory text in numbered sections. The explanatory text is an integral part of the standard, with the exception of the following items:

(1) in Section 13, examples of common usage of the elementary functions (under the heading Usage associated with each function);

(2) also in Section 13, notes (under the heading Notes associated with some of the functions); and

(3) examples and notes (labeled as such) presented at the end of any numbered section.

The word "may," as used in this document, consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could," or "might") are used to express ability, possibility, capacity, or consequentiality.

This proposal was prepared under the leadership of G. Myers with contributions by the following individuals, listed in alphabetical order: J. G. P. Barnes, W. J. Cody, P. M. Cohen, S. G. Cohen, K. W. Dritz, B. Ford, J. B. Goodenough, G. S. Hodgson, J. Kok, R. F. Mathis, T. G. Mattson, B. T. Smith, J. S. Squire, P. T. P. Tang, W. A. Whitaker, D. T. Winter, and M. Woodger. Many others contributed through international meetings and electronic mail reviews. Organizations lending support to this effort were the Naval Ocean Systems Center, Argonne National Laboratory, Westinghouse Electric Corporation, Numerical Algorithms Group, Centrum voor Wiskunde en Informatica, Software Productivity Consortium, Contel, Martin Marietta, the Software Engineering Institute of Carnegie–Mellon University, Quantitative Technology Corporation, and Alsys.

Bibliography

[1] M. Abramowitz and I. Stegun. Handbook of Mathematical Functions. U.S. Government Printing Office, Washington, D.C., 1964.

[2] W.J. Cody and W. Waite. Software Manual for the Elementary Functions, Prentice–Hall, 1980.

[3] B.Ford, J. Kok, and M.W. Rogers, editors. Scientific Ada, Cambridge University press, 1986.

[4] F.B. Hildebrand. Introduction to Numerical Analysis. McGraw–Hill, 1956.

[5] IEEE, IEEE Standard for Binary Floating–Point Arithmetic, ANSI/IEEE Std. 754–1985, IEEE, New York, 1985.

[6] IEEE, IEEE Standard for Radix–Independent Floating–Point Arithmetic. ANSI/IEEE Std. 854–1987, IEEE, New York, 1987.

[7] J. Kok. Proposal for Standard Mathematical Packages in Ada. CWI Report NM–R8718, Centrum voor Wiskunde en Informatica, Amsterdam, November 1987.

[8] R.F. Mathis. Elementary Functions Packages for Ada. In Proc. 1987 ACM SIGAda International Conference on the Ada Programming Language (Special issue of Ada letters), pages 95–100, December 1987.

[9] U.S. Department of Defense, Ada Joint Program Office, Reference Manual for the Ada Programming Language, ANSI/MIL–STD–1815A. U.S. Government Printing Office, Washington, D.C., 1983. Also adopted by ISO as ISO/8652–1987, Programming Languages–Ada.

Using the Math Library

The Math Library source is located in the **math** directory. To compile this source do the following:

1. Change to the **math** directory.
2. Run **newlib**.
3. Run **buildit.bat** to compile and link the Math Library source and a demo program called **tgef.a**. **Tgef.a** produces a test program called **testelem.exe**. **Testelem** demonstrates the generic elementary functions.

You have now created an Ada program library that you can link to from your own application program libraries.

You can modify and distribute this source as you like as long as you abide by the copyright restrictions documented in the source file **gef_.a**.

Specification

```
package ELEMENTARY_FUNCTIONS_EXCEPTIONS is
   ARGUMENT_ERROR : exception;
end ELEMENTARY_FUNCTIONS_EXCEPTIONS;

with ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <>;
package GENERIC_ELEMENTARY_FUNCTIONS is

   function SQRT     (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function LOG      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function LOG      (X, BASE   : FLOAT_TYPE)          return FLOAT_TYPE;
   function EXP      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function "**"     (X, Y      : FLOAT_TYPE)          return FLOAT_TYPE;

   function SIN      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function SIN      (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;
   function COS      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function COS      (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;
   function TAN      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function TAN      (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;
   function COT      (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function COT      (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;

   function ARCSIN   (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCSIN   (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCCOS   (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCCOS   (X, CYCLE  : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCTAN   (Y         : FLOAT_TYPE;
                      X         : FLOAT_TYPE := 1.0)   return FLOAT_TYPE;
   function ARCTAN   (Y         : FLOAT_TYPE;
                      X         : FLOAT_TYPE := 1.0;
                      CYCLE     : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCCOT   (X         : FLOAT_TYPE;
                      Y         : FLOAT_TYPE := 1.0)   return FLOAT_TYPE;
   function ARCCOT   (X         : FLOAT_TYPE;
                      Y         : FLOAT_TYPE := 1.0;
                      CYCLE     : FLOAT_TYPE)          return FLOAT_TYPE;

   function SINH     (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function COSH     (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function TANH     (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function COTH     (X         : FLOAT_TYPE)          return FLOAT_TYPE;

   function ARCSINH  (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCCOSH  (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCTANH  (X         : FLOAT_TYPE)          return FLOAT_TYPE;
   function ARCCOTH  (X         : FLOAT_TYPE)          return FLOAT_TYPE;

   ARGUMENT_ERROR : exception
            renames ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```

1. Purpose

This standard provides certain elementary mathematical functions. They were chosen because of their widespread utility in various application areas; moreover, they are needed to support general floating–point usage and to support generic packages for complex arithmetic and complex functions.

2. Functions provided

The following twenty mathematical functions are provided:

```
SQRT      LOG       EXP       "**"
SIN       COS       TAN       COT
ARCSIN    ARCCOS    ARCTAN    ARCCOT
SINH      COSH      TANH      COTH
ARCSINH   ARCCOSH   ARCTANH   ARCCOTH
```

These are the square root (SQRT), logarithm (LOG), and exponential (EXP) functions and the exponential operator (**); the circular trigonometric functions for sine (SIN), cosine (COS), tangent (TAN), and cotangent (COT) and their inverses (ARCSIN, ARCCOS, ARCTAN, and ARCCOT); and the hyperbolic trigonometric functions for sine (SINH), cosine (COSH), tangent (TANH), and cotangent (COTH) together with their inverses (ARCSINH, ARCCOSH, ARCTANH, and ARCCOTH).

3. Instantiations

This standard describes a generic package, GENERIC_ELEMENTARY_FUNCTIONS, which the user must instantiate to obtain a package. It has one generic formal parameter, which is a generic formal type named FLOAT_TYPE. At instantiation, the user must specify a floating–point subtype as the generic actual parameter to be associated with FLOAT_TYPE; it is referred to below as the "generic actual type." This type is used as the parameter and result type of the functions contained in the generic package.

The user is expressly allowed to specify a generic actual type having a range constraint. Such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when an argument outside the user's range is passed in a call to one of the functions, or when one of the functions attempts to return a value outside the user's range. Allowing the generic actual type to have a range constraint also has some implications for implementors (cf. Section 4).

In addition to the body of the generic package itself, implementors may provide (non–generic) library packages that can be used just like instantiations of the generic package for the predefined floating–point types. The name of a package serving as a replacement for an instantiation of GENERIC_ELEMENTARY_FUNCTIONS for the predefined type FLOAT should be ELEMENTARY_FUNCTIONS; for LONG_FLOAT and SHORT_FLOAT, the names should be LONG_ELEMENTARY_FUNCTIONS and SHORT_ELEMENTARY_FUNCTIONS, respectively; etc. When such a package is used in an application in lieu of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS, it must have the semantics implied by this standard for an instantiation of the generic package.

4. Implementations

Portable implementations of the body of GENERIC_ELEMENTARY_FUNCTIONS are strongly encouraged. However, implementations are not required to be portable. In particular, an implementation of the standard in Ada may use pragma INTERFACE or other pragmas, unchecked conversion, machine–code insertions, or other machine–dependent techniques as desired.

An implementation is allowed to limit the precision it supports (by stating an assumed maximum value for SYSTEM'MAX_DIGITS), since portable implementations would not, in general, be possible otherwise. An implementation is also allowed to make other reasonable assumptions about the environment in which it is to be used, but only when necessary in order to match algorithms to hardware characteristics in an economical

manner. All such limits and assumptions must be clearly documented. By convention, an implementation of GENERIC_ELEMENTARY_FUNCTIONS is said not to conform to this standard in any environment in which its limits or assumptions are not satisfied, and this standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

An implementation must function properly in a tasking environment. Apart from the obvious restriction that an implementation of GENERIC_ELEMENTARY_FUNCTIONS must avoid declaring variables that are global to the functions, no special constraints are imposed on implementations. Nothing in this standard requires the use of such global variables.

An implementation must not allow a range constraint to interfere with the internal computations of the functions, when the range constraint is included in the user's generic actual type. This places certain constraints on the way implementations can be designed.

Example:

Consider the following application code:

```
type MY_TYPE is digits 3 range 1.99 .. 4.00;
package MY_ELEMENTARY_FUNCTIONS is
   new GENERIC_ELEMENTARY_FUNCTIONS(MY_TYPE);
use MY_ELEMENTARY_FUNCTIONS;
Y : MY_TYPE;
     ...
Y := SQRT(4.0);
```

The internal computations of SQRT might use or generate numbers in the range 0.25 to 1.0, which are outside the range of MY_TYPE. Nevertheless, if the argument and result are within the range of the type, as they are in this example, then the implementation must return the result and must not raise an exception (such as CONSTRAINT_ERROR).

5. Exceptions

One exception, ARGUMENT_ERROR, is declared in GENERIC_ELEMENTARY_FUNCTIONS. This exception is raised by a function in the generic package only when the argument(s) of the function violate one or more of the conditions given in the function's domain definition (cf. Section 8). Note that these conditions are related only to the mathematical definition of the function and are therefore implementation independent.

The ARGUMENT_ERROR exception is declared as a renaming of the exception of the same name declared in the ELEMENTARY_FUNCTIONS_EXCEPTIONS package. Thus, this exception distinguishes neither between different kinds of argument errors, nor between different functions, nor between different instantiations of GENERIC_ELEMENTARY_FUNCTIONS.

Besides ARGUMENT_ERROR, the only exceptions allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS are predefined exceptions, as follows:

(1) Virtually any predefined exception is possible during the evaluation of an argument of a function in GENERIC_ELEMENTARY_FUNCTIONS. For example, NUMERIC_ERROR, CONSTRAINT_ERROR, or even PROGRAM_ERROR could be raised if an argument has an undefined value; and, as stated in Section 3, CONSTRAINT_ERROR will be raised when the value of an argument lies outside the range of the user's generic actual type. Additionally, STORAGE_ERROR could be raised, e.g.if insufficient storage is available to perform the call. All these exceptions are raised before the body of the function is entered and therefore have no bearing on implementations of GENERIC_ELEMENTARY_FUNCTIONS.

(2) Also as stated in Section 3, CONSTRAINT_ERROR will be raised when a function in GENERIC_ELEMENTARY_FUNCTIONS attempts to return a value outside the range of the user's generic actual type. The exception raised for this reason must be propagated to the caller of the function.

(3) Whenever the arguments of a function are such that a result permitted by the accuracy requirements would exceed FLOAT_TYPE'SAFE_LARGE in absolute value, as formalized below in Section 11, an implementation may raise (and must then propagate to the caller) the exception specified by Ada for signaling overflow.

(4) Once execution of the body of a function has begun, an implementation may propagate STORAGE_ERROR to the caller of the function, but only to signal the unexpected exhaustion of storage. Similarly, once execution of the body of a function has begun, an implementation may propagate PROGRAM_ERROR to the caller of the function, but only to signal errors made by the user of GENERIC_ELEMENTARY_FUNCTIONS.

No exception is allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS except those permitted by the foregoing rules. In particular, for arguments for which all results satisfying the accuracy requirements remain less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value, a function must locally handle an overflow occurring during the computation of an intermediate result, if such an overflow is possible, and not propagate an exception signaling that overflow to the caller of the function.

The only exceptions allowed during an instantiation of GENERIC_ELEMENTARY_FUNCTIONS, including the execution of the optional sequence of statements in the body of the instance, are STORAGE_ERROR and PROGRAM_ERROR, and then only for the reasons given in case (), above.

Note:

In the Ada Reference Manual, the exception specified for signaling overflow is NUMERIC_ERROR, but AI–00387 replaces that by CONSTRAINT_ERROR.

6. Arguments Outside the Range of Safe Numbers

The current Ada standard fails to define the result safe interval of any basic or predefined operation of a real subtype when the absolute value of one of its operands exceeds the largest safe number of the operand subtype. (The failure to define a result in this case occurs because no safe interval is defined for the operand in question.) In order to avoid imposing requirements that would, consequently, be more stringent than those of Ada itself, this standard likewise does not define the result of a contained function when the absolute value of one of its arguments exceeds FLOAT_TYPE'SAFE_LARGE. All of the accuracy requirements and other provisions of the following sections are understood to be implicitly qualified by the assumption that function arguments are less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value.

7. Method of Specification of Functions

Some of the functions have two overloaded forms. For each form of a function covered by this standard, the function is specified by its parameter and result type profile, the domain of its argument(s), its range, and the accuracy required of its implementation. The meaning of, and conventions applicable to, the domain, range, and accuracy specifications are described below.

8. Domain Definitions

The specification of each function covered by this standard includes, under the heading Domain, a characterization of the argument values for which the function is mathematically defined. It is expressed by inequalities or other conditions which the arguments must satisfy to be valid. The phrase "mathematically unbounded" in a domain definition indicates that all representable values of the argument are valid. Whenever the arguments fail to satisfy all the conditions, the implementation must raise ARGUMENT_ERROR. It must not raise that exception if all the conditions are satisfied.

Inability to deliver a result for valid arguments because the result overflows, for example, must not raise ARGUMENT_ERROR, but must be treated in the same way that Ada defines for its predefined floating–point operations (cf. Section 11); after all, one of these operations causes the overflow.

Note:

Unbounded portions of the domains of the functions EXP, "**", SINH, and COSH, which are "expansion" functions with unbounded or semi–unbounded mathematical domains, are unexploitable because the corresponding function values (satisfying the accuracy requirements) can not be represented. Their "usable domains," i.e.the portions of the mathematical domains given in their domain definitions that are exploitable in the sense that they produce representable results, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these usable domains can only be stated approximately. In a similar manner, functions such as TAN and COT with periodic "poles" in their domains might or might not (depending on the implementation) have small unusable portions of their domains in the vicinities of the poles. Also, range constraints in the user's generic actual type can, by narrowing a function's range, make further portions of the function's domain unusable.

9. Range Definitions

The usual mathematical meaning of the "range" of a function is the set of values into which the function maps the values in its domain. Some of the functions covered by this standard (for example, ARCSIN) are mathematically multivalued, in the sense that a given argument value can be mapped by the function into many different result values. By means of range restrictions, this standard imposes a uniqueness requirement on the results of multivalued functions, thereby reducing them to single–valued functions.

Some of the functions covered by this standard (for example, EXP) have asymptotic behavior for extremely positive or negative arguments. Although there is no finite argument for which such a function can mathematically yield its asymptotic limit, that limit is always included in its range here, and it is an allowed result of the implemented function, in recognition of the fact that the limit value itself could be closer to the mathematical result than any other representable value.

The range of each function is shown under the heading Range in the specifications. Range definitions take the form of inequalities limiting the function value. An implementation must not exceed a limit of the range when that limit is a safe number of FLOAT_TYPE (like 0.0, 1.0, or CYCLE/4.0 for certain values of CYCLE). On the other hand, when a range limit is not a safe number of FLOAT_TYPE (like Pi, or CYCLE/4.0 for certain other values of CYCLE), an implementation is allowed to exceed the range limit, but it is not allowed to exceed the safe number of FLOAT_TYPE next beyond the range limit in the direction away from the interior of the range; this is in general the best that can be expected from a portable implementation. Effectively, therefore, range definitions have the added effect of imposing accuracy requirements on implementations above and beyond those presented under the heading Accuracy in the specifications (cf. Section 10).

The phrase "mathematically unbounded" in a range definition indicates that the range of values of the function is not bounded by its mathematical definition. It also implies that the function is not mathematically multivalued.

Note:

Unbounded portions of the ranges of the functions SQRT, LOG, ARCSINH, and ARCCOSH, which are "contraction" functions with unbounded or semi–unbounded mathematical ranges, are unreachable because the corresponding arguments can not be represented. Their "reachable ranges," i.e.the portions of the mathematical ranges given in their range definitions that are reachable through appropriate arguments, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these reachable ranges can only be stated approximately. Also, range constraints in the user's generic actual type can, by narrowing a function's domain, make further portions of the function's range unreachable.

10. Accuracy Requirements

Because they are implemented on digital computers with only finite precision, the functions provided in this generic package can, at best, only approximate the corresponding mathematically defined functions.

The accuracy requirements contained in this standard define the latitude that implementations are allowed in approximating the intended precise mathematical result with floating–point computations. Accuracy re-

quirements of two kinds are stated under the heading Accuracy in the specifications. Additionally, range definitions stated under the heading Range impose requirements that constrain the values implementations may yield, so the range definitions are another source of accuracy requirements (the precise meaning of a range limit that is not a safe number of FLOAT_TYPE, as an accuracy requirement, is discussed above in Section 9). Every result yielded by a function is subject to all of the function's applicable accuracy requirements, except in the one case described in Section 12, below. In that case, the result will satisfy a small absolute error requirement in lieu of the other accuracy requirements defined for the function.

The first kind of accuracy requirement used under the heading Accuracy in the specifications is a bound on the relative error in the computed value of the function, which must hold (except as provided by the rules in Sections 11 and 12) for all arguments satisfying the conditions in the domain definition, providing the mathematical result is non–zero. For a given function f, the relative error re(X) in a computed result F(X) at the argument X is defined in the usual way,

$$re(X) = \left| \frac{F(X) - f(X)}{f(X)} \right|$$

providing the mathematical result f(X) is non–zero. (The relative error is not defined when the mathematical result is zero.) For each function, the bound on the relative error is identified under the heading Accuracy as its maximum relative error.

The second kind of accuracy requirement used under the heading Accuracy in the specifications is a stipulation, in the form of an equality, that the implementation must deliver "prescribed results" for certain special arguments. It is used for two purposes: to define the computed result to be zero when the relative error is undefined, i.e., when the mathematical result is zero, and to strengthen the accuracy requirements at special argument values. When such a prescribed result is a safe number of FLOAT_TYPE (like 0.0, 1.0, or CYCLE/4.0 for certain values of CYCLE), an implementation must deliver that result. On the other hand, when a prescribed result is not a safe number of FLOAT_TYPE (like Pi, or CYCLE/4 for certain other values of CYCLE), an implementation may deliver any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them.

Range definitions, under the heading Range in the specifications, are an additional source of accuracy requirements, as stated above in Section 9. As an accuracy requirement, a range definition (other than "mathematically unbounded") has the effect of eliminating some of the values permitted by the maximum relative error requirements, e.g., those outside the range.

11. Overflow

Floating–point hardware is typically incapable of representing numbers whose absolute value exceeds some implementation–defined maximum. For the type FLOAT_TYPE, that maximum will be at least FLOAT_TYPE'SAFE_LARGE. For the functions defined by this standard, whenever the maximum relative error requirements permit a result whose absolute value is greater than FLOAT_TYPE'SAFE_LARGE, the implementation may (1) yield any result permitted by the maximum relative error requirements, or (2) raise the exception specified by Ada for signaling overflow.

Notes:

This rule permits an implementation to raise an exception, instead of delivering a result, for arguments for which the mathematical result is close to but does not exceed FLOAT_TYPE'SAFE_LARGE in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result does exceed FLOAT_TYPE'SAFE_LARGE in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

The rule is motivated by the behavior prescribed by the Ada Reference Manual for the predefined operations. That is, when the set of possible results of a predefined operation includes a number whose absolute value exceeds the implementation–defined maximum, the implementation is allowed to raise the exception specified for signaling overflow instead of delivering a result.

In the Ada Reference Manual, the exception specified for signaling overflow is NUMERIC_ERROR, but AI–00387 replaces that by CONSTRAINT_ERROR.

12. Underflow

Floating–point hardware is typically incapable of representing non–zero numbers whose absolute value is less than some implementation–defined minimum. For the type FLOAT_TYPE, that minimum will be at most FLOAT_TYPE'SAFE_SMALL. For the functions defined by this standard, whenever the maximum relative error requirements permit a result whose absolute value is less than FLOAT_TYPE'SAFE_SMALL and a prescribed result is not stipulated, the implementation may (1) yield any result permitted by the maximum relative error requirements; (2) yield any non–zero result having the correct sign and an absolute value less than or equal to FLOAT_TYPE'SAFE_SMALL; or (3) yield zero.

Notes:

Whenever part (2) or (3) of this rule takes effect, the maximum relative error requirements are, in general, unachievable and are waived. In such cases, the computed result will exhibit an error which, while not necessarily small in relative terms, is small in absolute terms. The absolute error will, in these cases, be less than or equal to FLOAT_TYPE'SAFE_SMALL/(1.0 – mre), where mre is the maximum relative error specified for the function under the heading Accuracy.

The rule permits an implementation to deliver a result violating the maximum relative error requirements for arguments for which the mathematical result equals or slightly exceeds FLOAT_TYPE'SAFE_SMALL in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result is less than FLOAT_TYPE'SAFE_SMALL in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

The rule is motivated by the behavior prescribed by the Ada Reference Manual for predefined operations. That is, when the set of possible results of a predefined operation includes a non–zero number whose absolute value is less than the implementation–defined minimum, the implementation is allowed to yield zero or any non–zero number having the correct sign and an absolute value less than or equal to that minimum. An exception is never raised in this case.

13. Specifications of the Functions

13.1 SQRT

```
Declaration:
  function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  SQRT(X) ~ sqrt(X)
Usage:
  Z := SQRT(X);
Domain:
  X >= 0.0
Range:
  SQRT(X) >= 0.0
Accuracy:
   (a) Maximum relative error = 2.0 * FLOAT_TYPE'BASE'EPSILON
   (b) SQRT(0.0) = 0.0
Notes:
   (a) The upper bound of the reachable range of SQRT is approximately
   given by SQRT(X)<=sqrt(FLOAT_TYPE'SAFE_LARGE)
   (b) Other standards might impose additional constraints on SQRT. For
   example, the IEEE standards for binary and radix-independent
   floating-point arithmetic require greater accuracy in the result of
   SQRT than this standard requires, and they require that
   SQRT(-0.0)=-0.0.
   An implementation of SQRT in GENERIC_ELEMENTARY_FUNCTIONS that
   conforms to this standard will conform to those other standards if
   it satisfies their additional constraints.
```

13.2 LOG (natural base)

```
Declaration:
  function LOG (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  LOG(X) ~ ln(X)
Usage:
  Z := LOG(X);    -- natural logarithm
Domain:
  X > 0.0
Range:
  Mathematically unbounded
Accuracy:
   (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
   (b) LOG(1.0) = 0.0
Notes:
  The reachable range of LOG is approximately given by
  ln(FLOAT_TYPE'SAFE_SMALL) <= LOG(X) <= ln(FLOAT_TYPE'SAFE_LARGE)
```

13.3 LOG (arbitrary base)

```
Declaration:
  function LOG (X, BASE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  LOG(X,BASE) ~ log to base BASE of X
Usage:
  Z := LOG(X, 10.0);    -- base 10 logarithm
  Z := LOG(X, 2.0);     -- base 2 logarithm
  Z := LOG(X, BASE);    -- base BASE logarithm
Domain:
  (a) X > 0.0
  (b) BASE > 0.0
  (c) BASE /= 1.0
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) LOG(1.0,BASE) = 0.0
Notes:
  (a) When BASE > 1.0, the reachable range of LOG is approximately
  given by
      log to base BASE of FLOAT_TYPE'SAFE_SMALL <= LOG(X,BASE) <=
      log to base BASE of FLOAT_TYPE'SAFE_LARGE
  (b) When 0.0 < BASE < 1.0, the reachable range of LOG is
  approximately given by
      log to base BASE of FLOAT_TYPE'SAFE_LARGE <= LOG(X,BASE) <=
      log to base BASE of FLOAT_TYPE'SAFE_SMALL
```

13.4 EXP

```
Declaration:
  function EXP (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  EXP(X) ~ e raised to the X power
Usage:
  Z := EXP(X);    -- e raised to the power X
Domain:
  Mathematically unbounded
Range:
  EXP(X) >= 0.0
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) EXP(0.0) = 1.0
Notes:
  The usable domain of EXP is approximately given by
  X <= ln(FLOAT_TYPE'SAFE_LARGE).
```

13.5 "**"

```
Declaration:
  function "**" (X, Y : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  X ** Y ~ X raised to the power Y
Usage:
  Z := X ** Y;    -- X raised to the power Y
Domain:
  (a) X >= 0.0
  (b) Y > 0.0 when X = 0.0
Range:
  X ** Y >= 0.0
Accuracy:
  (a) Maximum relative error (when X > 0.0) =
  (4.0+|Y*ln(X)|/32.0) * FLOAT_TYPE'BASE'EPSILON
  (b) X ** 0.0 = 1.0 when X > 0.0
  (c) 0.0 ** Y = 0.0 when Y > 0.0
  (d) X ** 1.0 = X
  (e) 1.0 ** Y =1.0
Notes:
  The usable domain of "**", when X > 0.0, is approximately the set
  of values for X and Y satisfying
  Y*ln(X) <= ln(FLOAT_TYPE'SAFE_LARGE)
  This imposes a positive upper bound on Y (as a function of X) when
  X > 1.0, and a negative lower bound on Y (as a function of X) when
  0.0 < X < 1.0.
```

13.6 SIN (natural cycle)

```
Declaration:
  function SIN (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  SIN(X) ~ sin(X)
Usage:
  Z := SIN(X);    -- X in radians
Domain:
  Mathematically unbounded
Range:
  |SIN(X)| <= 1.0
Accuracy:
  (a) Maximum relative error = 2.0 * FLOAT_TYPE'BASE'EPSILON
  when |X| is less than or equal to some documented implementation-
  dependent threshold, which must not be less than
    FLOAT_TYPE'MACHINE_RADIX ** (FLOAT_TYPE'MACHINE_MANTISSA/2)
  For larger values of |X|, degraded accuracy is allowed. An
  implementation must document its behavior for large |X|.
  (b) SIN(0.0) = 0.0
```

13.7 SIN (arbitrary cycle)

```
Declaration:
  function SIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  SIN(X,CYCLE) ~ sin(2Pi * X/CYCLE)
Usage:
  Z := SIN(X, 360.0);    -- X in degrees
  Z := SIN(X, 1.0);      -- X in bams (binary angular measure)
  Z := SIN(X, CYCLE);    -- X in units such that one complete
                         -- cycle of rotation corresponds to
                         -- X = CYCLE
Domain:
  (a) X mathematically unbounded
  (b) CYCLE > 0.0
Range:
  |SIN(X,CYCLE)| <= 1.0
Accuracy:
  (a) Maximum relative error = 2.0 * FLOAT_TYPE'BASE'EPSILON
  (b) For integer k, SIN(X,CYCLE)= 0.0 when X=k*CYCLE/2.0
                                   1.0 when X=(4k+1)*CYCLE/4.0
                                   -1.0 when X=(4k+3)*CYCLE/4.0
```

13.8 COS (natural cycle)

```
Declaration:
  function COS (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COS(X) ~ cos(X)
Usage:
  Z := COS(X);    -- X in radians
Domain:
  Mathematically unbounded
Range:
  |COS(X)| <= 1.0
Accuracy:
  (a) Maximum relative error = 2.0 * FLOAT_TYPE'BASE'EPSILON
  when |X| is less than or equal to some documented implementation-
  dependent threshold, which must not be less than
    FLOAT_TYPE'MACHINE_RADIX ** (FLOAT_TYPE'MACHINE_MANTISSA/2)
  For larger values of |X|, degraded accuracy is allowed. An
  implementation must document its behavior for large |X|.
  (b) COS(0.0) = 1.0
```

13.9 COS (arbitrary cycle)

```
Declaration:
  function COS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COS(X,CYCLE) ~ cos(2Pi*X/CYCLE)
Usage:
  Z := COS(X, 360.0);    -- X in degrees
  Z := COS(X, 1.0);      -- X in bams
  Z := COS(X, CYCLE);    -- X in units such that one complete
                         -- cycle of rotation corresponds to
                         -- X = CYCLE
Domain:
  (a) X mathematically unbounded
  (b) CYCLE > 0.0
Range:
  |COS(X,CYCLE)| <= 1.0
Accuracy:
  (a) Maximum relative error = 2.0 * FLOAT_TYPE'BASE'EPSILON
  (b) For integer k, COS(X,CYCLE) = 1.0 when X=k*CYCLE
                                    0.0 when X=(2k+1)*CYCLE/4.0
                                    -1.0 when X=(2k+1)*CYCLE/2.0
```

13.10 TAN (natural cycle)

```
Declaration:
  function TAN (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  TAN(X) ~ tan(X)
Usage:
  Z := TAN(X);    -- X in radians
Domain:
  Mathematically unbounded
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  when |X| is less than or equal to some documented implementation-
  dependent threshold, which must not be less than
    FLOAT_TYPE'MACHINE_RADIX ** (FLOAT_TYPE'MACHINE_MANTISSA/2)
  For larger values of |X|, degraded accuracy is allowed. An
  implementation must document its behavior for large |X|.
  (b) TAN(0.0) = 0.0
```

13.11 TAN (arbitrary cycle)

```
Declaration:
  function TAN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  TAN(X,CYCLE) ~ tan(2Pi*X/CYCLE)
Usage:
  Z := TAN(X, 360.0);    -- X in degrees
  Z := TAN(X, 1.0);      -- X in bams
  Z := TAN(X, CYCLE);    -- X in units such that one complete
                         -- cycle of rotation corresponds to
                         -- X = CYCLE
Domain:
  (a) X /= (2k+1)*CYCLE/4.0, for integer k
  (b) CYCLE > 0.0
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) TAN(X,CYCLE) = 0.0 when X=k*CYCLE/2.0, for integer k
```

13.12 COT (natural cycle)

```
Declaration:
  function COT (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COT(X) ~ cot(X)
Usage:
  Z := COT(X);    -- X in radians
Domain:
  X /= 0.0
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  when |X| is less than or equal to some documented implementation-
  dependent threshold, which must not be less than
    FLOAT_TYPE'MACHINE_RADIX ** (FLOAT_TYPE'MACHINE_MANTISSA/2)
  For larger values of |X|, degraded accuracy is allowed. An
  implementation must document its behavior for large |X|.
```

13.13 COT (arbitrary cycle)

```
Declaration:
  function COT (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COT(X,CYCLE) ~ cot(2Pi*X/CYCLE)
Usage:
  Z := COT(X, 360.0);    -- X in degrees
  Z := COT(X, 1.0);      -- X in bams
  Z := COT(X, CYCLE);    -- X in units such that one complete
                         -- cycle of rotation corresponds to
                         -- X = CYCLE
Domain:
  (a) X /= k*CYCLE/2.0, for integer k
  (b) CYCLE > 0.0
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) COT(X,CYCLE) = 0.0 when X=(2k+1)*CYCLE/4.0, for integer k
```

13.14 ARCSIN (natural cycle)

```
Declaration:
  function ARCSIN (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCSIN(X) ~ arcsine(X)
Usage:
  Z := ARCSIN(X);    -- Z in radians
Domain:
  |X| <= 1.0
Range:
  |ARCSIN(X)| <= Pi/2
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCSIN(0.0) = 0.0
  (c) ARCSIN(1.0) = Pi/2
  (d) ARCSIN(-1.0) = -Pi/2
Notes:
  - Pi/2 and Pi/2 are not safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (c) or (d) applies, an implementation may
  approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.15 ARCSIN (arbitrary cycle)

```
Declaration:
  function ARCSIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCSIN(X,CYCLE) ~ arcsin(X)*CYCLE/2Pi
Usage:
  Z := ARCSIN(X, 360.0);    -- Z in degrees
  Z := ARCSIN(X, 1.0);      -- Z in bams
  Z := ARCSIN(X, CYCLE);    -- Z in units such that one complete
                            -- cycle of rotation corresponds to
                            -- Z = CYCLE
Domain:
  (a) |X| <= 1.0
  (b) CYCLE > 0.0
Range:
  |ARCSIN(X,CYCLE) <= CYCLE/4.0
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCSIN(0.0,CYCLE) = 0.0
  (c) ARCSIN(1.0,CYCLE) = CYCLE/4.0
  (d) ARCSIN(-1.0,CYCLE) = -CYCLE/4.0
Notes:
  - CYCLE/4.0 and CYCLE/4.0
  might not be safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (c) or (d) applies, an implementation may
  approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.16 ARCCOS (natural cycle)

```
Declaration:
  function ARCCOS (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCCOS(X) ~ arccos(X)
Usage:
  Z := ARCCOS(X);    -- Z in radians
Domain:
  |X| <= 1.0
Range:
  0.0 <= ARCCOS(X) <= Pi
Accuracy:
  (a)  Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCCOS(1.0) = 0.0
  (c) ARCCOS(0.0) = Pi/2
  (d) ARCCOS(-1.0) = Pi
Notes:
  Pi/2 and Pi are not safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (c) or (d) applies, an implementation may
  approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.17 ARCCOS (arbitrary cycle)

```
Declaration:
  function ARCCOS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCCOS(X,CYCLE) ~ arccos(X)*CYCLE/2Pi
Usage:
  Z := ARCCOS(X, 360.0);    -- Z in degrees
  Z := ARCCOS(X, 1.0);      -- Z in bams
  Z := ARCCOS(X, CYCLE);    -- Z in units such that one complete
                            -- cycle of rotation corresponds to
                            -- Z = CYCLE
Domain:
  (a) |X| <= 1.0
  (b) CYCLE > 0.0
Range:
  0.0 <= ARCCOS(X,CYCLE) <= CYCLE/2.0
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCCOS(1.0,CYCLE) = 0.0
  (c) ARCCOS(0.0,CYCLE) = CYCLE/4.0
  (d) ARCCOS(-1.0,CYCLE) = CYCLE/2.0
Notes:
  CYCLE/4.0 and CYCLE/2.0
  might not be safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (c) or (d) applies, an implementation may
  approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.18 ARCTAN (natural cycle)

```
Declaration:
  function ARCTAN (Y : FLOAT_TYPE;
                   X : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
Definition:
  (a) ARCTAN(Y) ~ arctan(Y)
  (b) ARCTAN(Y,X) ~ arctan(Y/X) when X >= 0.0
                    arctan(Y/X)+Pi when X < 0.0 and Y >= 0.0
                    arctan(Y/X)-Pi when X < 0.0 and Y < 0.0
Usage:
  Z := ARCTAN(Y);       -- Z, in radians, is the angle (in the
                        -- quadrant containing the point (1.0,Y))
                        -- whose tangent is Y
  Z := ARCTAN(Y, X);    -- Z, in radians, is the angle (in the
                        -- quadrant containing the point (X,Y))
                        -- whose tangent is Y/X
Domain:
  X /= 0.0 when Y = 0.0
Range:
    (a) |ARCTAN(Y)| <= Pi/2
    (b) 0.0 < ARCTAN(Y,X) <= Pi when Y >= 0.0
    (c) -Pi <= ARCTAN(Y,X) <= 0.0 when Y < 0.0
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCTAN(0.0) = 0.0
  (c) ARCTAN((0.0,X) = 0.0 when X > 0.0
                       Pi when X < 0.0
  (d) ARCTAN(Y,0.0) = Pi/2 when Y > 0.0
                     -Pi/2 when Y < 0.0
Notes:
  -Pi,-Pi/2,Pi/2 and Pi are not safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (c) or (d) applies, an implementation may
  approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.19 ARCTAN (arbitrary cycle)

```
Declaration:
  function ARCTAN (Y     : FLOAT_TYPE;
                   X     : FLOAT_TYPE := 1.0;
                   CYCLE : FLOAT_TYPE)        return FLOAT_TYPE;
Definition:
  (a) ARCTAN(Y,CYCLE)   ~ arctan(Y)*CYCLE/2Pi
  (b) ARCTAN(Y,X,CYCLE)
            ~ arctan(Y/X)*CYCLE/2Pi when X >= 0.0
              (arctan(Y/X)+Pi)*CYCLE/2Pi when X < 0.0 and Y >= 0.0
              (arctan(Y/X)-Pi)*CYCLE/2Pi when X < 0.0 and Y < 0.0
Usage:
  Z := ARCTAN(Y, CYCLE => 360.0);   -- Z, in degrees, is the
                                    -- angle (in the quadrant
                                    -- containing the point
```

```
                                        -- (1.0,Y)) whose tangent is Y
   Z := ARCTAN(Y, CYCLE => 1.0);        -- Z, in bams, is the
                                        -- angle (in the quadrant
                                        -- containing the point
                                        -- (1.0,Y)) whose tangent is Y
   Z := ARCTAN(Y, CYCLE => CYCLE);      -- Z, in units such that one
                                        -- complete cycle of rotation
                                        -- corresponds to Z = CYCLE,
                                        -- is the angle (in the
                                        -- quadrant containing the
                                        -- point (1.0,Y)) whose
                                        -- tangent is Y
   Z := ARCTAN(Y, X, 360.0);            -- Z, in degrees, is the
                                        -- angle (in the quadrant
                                        -- containing the point (X,Y))
                                        -- whose tangent is Y/X
   Z := ARCTAN(Y, X, 1.0);              -- Z, in bams, is the
                                        -- angle (in the quadrant
                                        -- containing the point (X,Y))
                                        -- whose tangent is Y/X
   Z := ARCTAN(Y, X, CYCLE);            -- Z, in units such that one
                                        -- complete cycle of rotation
                                        -- corresponds to Z = CYCLE,
                                        -- is the angle (in the
                                        -- quadrant containing the
                                        -- point (X,Y)) whose
                                        -- tangent is Y/X
```

Domain:

(a) X /= 0.0 when Y = 0.0

(b) CYCLE > 0.0

Range:

(a) |ARCTAN(Y,CYCLE)| <= CYCLE/4.0

(b) 0.0 <= ARCTAN(Y,X,CYCLE) <= CYCLE/2.0 when Y >= 0.0

(c) -CYCLE/2.0 <= ARCTAN(Y,X,CYCLE) <= 0.0 when Y < 0.0

Accuracy:

(a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON

(b) ARCTAN(0.0,CYCLE) = 0.0

(c) ARCTAN(0.0,X,CYCLE) = 0.0 when X > 0.0
CYCLE/2.0 when X < 0.0

(d) ARCTAN(Y,0.0,CYCLE) = CYCLE/4.0 when Y > 0.0
-CYCLE/4.0 when Y < 0.0

Notes:

-CYCLE/2.0, -CYCLE/4.0, CYCLE/4.0 and CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.20 ARCCOT (natural cycle)

```
Declaration:
  function ARCCOT (X : FLOAT_TYPE;
                   Y : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
Definition:
  (a) ARCCOT(X) ~ arccot(X)
  (b) ARCCOT(X,Y) ~ arccot(X/Y) when Y >= 0.0
                    arccot(X/Y)-Pi when Y < 0.0
Usage:
  Z := ARCCOT(X);       -- Z, in radians, is the angle (in the
                        -- quadrant containing the point (X,1.0)
                        -- whose cotangent is X
  Z := ARCCOT(X, Y);    -- Z, in radians, is the angle (in the
                        -- quadrant containing the point (X,Y))
                        -- whose cotangent is X/Y
Domain:
  Y /= 0.0 when X = 0.0
Range:
  (a) 0.0 <= ARCCOT(X) <= Pi
  (b) 0.0 <= ARCCOT(X,Y) <= Pi when Y >= 0.0
  (c) -Pi <= ARCCOT(X,Y) <= 0.0 when Y < 0.0
Accuracy:
  (a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCCOT(0.0) = Pi/2
  (c) ARCCOT(0.0,Y) = Pi/2 when Y > 0.0
                      -Pi/2 when Y < 0.0
  (d) ARCCOT(X,0.0) = 0.0 when X > 0.0
                      Pi when X < 0.0
Notes:
  -Pi,-Pi/2,Pi/2 and Pi are not safe numbers of FLOAT_TYPE. Accordingly,
  an implementation may exceed the range limits, but only slightly;
  cf. Section 9 for a precise statement of the requirements. Similarly,
  when accuracy requirement (b), (c), or (d) applies, an implementation
  may approximate the prescribed result, but only within narrow limits;
  cf. Section 10 for a precise statement of the requirements.
```

13.21 ARCCOT (arbitrary cycle)

```
Declaration:
  function ARCCOT (X     : FLOAT_TYPE;
                   Y     : FLOAT_TYPE := 1.0;
                   CYCLE : FLOAT_TYPE)        return FLOAT_TYPE;
Definition:
  (a) ARCCOT(X,CYCLE) ~ arccot(X)*CYCLE/2Pi
  (b) ARCCOT(X,Y) ~ arccot(X/Y)*CYCLE/2Pi when  Y >= 0.0
                    (arccot(X/Y)-Pi)*CYCLE/2Pi  Y < 0.0
Usage:
  Z := ARCCOT(X, CYCLE => 360.0);   -- Z, in degrees, is the
                                    -- angle (in the quadrant
                                    -- containing the point
                                    -- (X,1.0)) whose cotangent
                                    -- is X
  Z := ARCCOT(X, CYCLE => 1.0);     -- Z, in bams, is the
```

```
                                        -- angle (in the quadrant
                                        -- containing the point
                                        -- (X,1.0)) whose cotangent
                                        -- is X
Z := ARCCOT(X, CYCLE => CYCLE);         -- Z, in units such that one
                                        -- complete cycle of rotation
                                        -- corresponds to Z = CYCLE,
                                        -- is the angle (in the
                                        -- quadrant containing the
                                        -- point (X,1.0)) whose
                                        -- cotangent is X
Z := ARCCOT(X, Y, 360.0);               -- Z, in degrees, is the
                                        -- angle (in the quadrant
                                        -- containing the point (X,Y))
                                        -- whose cotangent is X/Y
Z := ARCCOT(X, Y, 1.0);                 -- Z, in bams, is the
                                        -- angle (in the quadrant
                                        -- containing the point (X,Y)
                                        -- whose cotangent is X/Y
Z := ARCCOT(X, Y, CYCLE);               -- Z, in units such that one
                                        -- complete cycle of rotation
                                        -- corresponds to Z = CYCLE
                                        -- is the angle (in the
                                        -- quadrant containing the
                                        -- point (X,Y)) whose
                                        -- cotangent is X/Y
```

Domain:

(a) Y /= 0.0 when X = 0.0

(b) CYCLE > 0.0

Range:

(a) 0.0 <= ARCCOT(X,CYCLE=>CYCLE) <= CYCLE/2.0

(b) 0.0 <= ARCCOT(X,Y,CYCLE) <= CYCLE/2.0 when Y >= 0.0

(c) -CYCLE/2.0 <= ARCCOT(X,Y,CYCLE) <= 0.0 when Y < 0.0

Accuracy:

(a) Maximum relative error = 4.0 * FLOAT_TYPE'BASE'EPSILON

(b) ARCCOT(0.0,CYCLE=>CYCLE) = CYCLE/4.0

(c) ARCCOT(0.0,Y,CYCLE) = CYCLE/4.0 when Y > 0.0
-CYCLE/4.0 when Y < 0.0

(d) ARCCOT(X,0.0,CYCLE) = 0.0 when X > 0.0
CYCLE/2.0 when X < 0.0

Notes:

- CYCLE/2.0, - CYCLE/4.0, CYCLE/4.0 and CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (b), (c), or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.22 SINH

```
Declaration:
  function SINH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  SINH(X) ~ sinh X
Usage:
  Z := SINH(X);
Domain:
  Mathematically unbounded
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) SINH(0.0) = 0.0
Notes:
  The usable domain of SINH is approximately given by
  |X| <= ln(FLOAT_TYPE'SAFE_LARGE)+ln(2.0)
```

13.23 COSH

```
Declaration:
  function COSH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COSH(X) ~ cosh X
Usage:
  Z := COSH(X);
Domain:
  Mathematically unbounded
Range:
  COSH(X) >= 1.0
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) COSH(0.0) = 1.0
Notes:
  The usable domain of COSH is approximately given by
  |X| <= ln(FLOAT_TYPE'SAFE_LARGE)+ln(2.0)
```

13.24 TANH

```
Declaration:
  function TANH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  TANH(X) ~ tanh X
Usage:
  Z := TANH(X);
Domain:
  Mathematically unbounded
Range:
  |TANH(X)| <= 1.0
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) TANH(0.0) = 0.0
```

13.25 COTH

```
Declaration:
  function COTH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  COTH(X) ~ coth X
Usage:
  Z := COTH(X);
Domain:
  X /= 0.0
Range:
  |COTH(X)| >= 1.0
Accuracy:
  Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
```

13.26 ARCSINH

```
Declaration:
  function ARCSINH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCSINH(X) ~ arcsinh X
Usage:
  Z := ARCSINH(X);
Domain:
  Mathematically unbounded
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCSINH(0.0) = 0.0
Notes:
  The reachable range of ARCSINH is approximately given by
    |ARCSINH(X)| <= ln(FLOAT_TYPE'SAFE_LARGE)+ln(2.0)
```

13.27 ARCCOSH

```
Declaration:
  function ARCCOSH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCCOSH(X) ~ arccosh X
Usage:
  Z := ARCCOSH(X);
Domain:
  X >= 1.0
Range:
  ARCCOSH(X) >= 0.0
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCCOSH(1.0) = 0.0
Notes:
  The upper bound of the reachable range of ARCCOSH is approximately
  given by  ARCCOSH(X) <= ln(FLOAT_TYPE'SAFE_LARGE)+ln(2.0)
```

13.28 ARCTANH

```
Declaration:
  function ARCTANH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCTANH(X) ~ arctanh X
Usage:
  Z := ARCTANH(X);
Domain:
  |X| < 1.0
Range:
  Mathematically unbounded
Accuracy:
  (a) Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
  (b) ARCTANH(0.0) = 0.0
```

13.29 ARCCOTH

```
Declaration:
  function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE;
Definition:
  ARCCOTH(X) ~ arccoth X
Usage:
  Z := ARCCOTH(X);
Domain:
  |X| > 1.0
Range:
  Mathematically unbounded
Accuracy:
  Maximum relative error = 8.0 * FLOAT_TYPE'BASE'EPSILON
```